Recursive Objects
Class outline:

- Linked Lists
- Link class
- Tree class
Linked lists
Why do we need a new list?

Python lists are implemented as a "dynamic array", which isn't optimal for all use cases.

-inserting an element is slow, especially near front of list:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3300</td>
<td>3301</td>
<td>3302</td>
<td>3303</td>
<td>3304</td>
<td>3305</td>
<td></td>
</tr>
</tbody>
</table>

What should we insert?

value: Z @ index: 3 ➔ Insert
Why do we need a new list?

Python lists are implemented as a "dynamic array", which isn't optimal for all use cases.

-insert- Inserting an element is slow, especially near front of list:

<table>
<thead>
<tr>
<th>&quot;A&quot;</th>
<th>&quot;B&quot;</th>
<th>&quot;C&quot;</th>
<th>&quot;D&quot;</th>
<th>&quot;E&quot;</th>
<th>&quot;F&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3300</td>
<td>3301</td>
<td>3302</td>
<td>3303</td>
<td>3304</td>
<td>3305</td>
</tr>
</tbody>
</table>

What should we insert?

value: Z @ index: 3 Insert

- Sometimes inserting too many elements can require re-creating the entire list in memory, if it exceeds the pre-allocated memory.
Linked lists

A linked list is a chain of objects where each object holds a value and a reference to the next link. The list ends when the final reference is empty.

What should we insert?

value: Z @ index: 5 Insert
Linked lists

A linked list is a chain of objects where each object holds a **value** and a **reference to the next link**. The list ends when the final reference is empty.

What should we insert?

value: **Z** @ index: **5**  

Linked lists require more space but provide faster insertion.
A Link class

class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

How would we use that?
A Link class

class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

How would we use that?

ll = Link("A", Link("B", Link("C")))

Try in PythonTutor
A fancier LinkedList

class Link:
    """A linked list."""
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '  
            self = self.rest
        return string + str(self.first) + '>

It's built-in to code.cs61a.org and you can draw() any Link.
Creating linked lists
Creating a range

Similar to: `[x for x in range(3, 6)]`

```python
def range_link(start, end):
    """Return a Link containing consecutive integers from START to END, not including END.
    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """
```

Try in PythonTutor
Creating a range

Similar to `[x for x in range(3, 6)]

def range_link(start, end):
    """Return a Link containing consecutive integers from START to END, not including END."
    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """
    if start >= end:
        return Link.empty
    return Link(start, range_link(start + 1, end))

Try in PythonTutor
Exercise: Mapping a linked list

Similar to \([f(x) \text{ for } x \text{ in } \text{lst}]\)

```python
def map_link(f, ll):
    """Return a Link that contains f(x) for each x in Link LL."
    >>> square = lambda x: x * x
    >>> map_link(square, range_link(3, 6))
    Link(9, Link(16, Link(25)))
    """
```

Try in PythonTutor
Exercise: Mapping a linked list (Solution)

Similar to \([f(x) \text{ for } x \text{ in } \text{lst}]\)

```python
def map_link(f, ll):
    """Return a Link that contains \(f(x)\) for each \(x\) in Link LL."
    >>> square = lambda x: x * x
    >>> map_link(square, range_link(3, 6))
    Link(9, Link(16, Link(25)))
    """
    if ll is Link.empty:
        return Link.empty
    return Link(f(ll.first), map_link(f, ll.rest))
```

Try in PythonTutor
Exercise: Filtering a linked list

Similar to \[ x \text{ for } x \text{ in } \text{lst} \text{ if } f(x) \]  

```python
def filter_link(f, ll):
    """Return a Link that contains only the elements x of Link LL for which f(x) is a true value.
    >>> is_odd = lambda x: x % 2 == 1
    >>> filter_link(is_odd, range_link(3, 6))
    Link(3, Link(5))
    """
```

Try in PythonTutor
Exercise: Filtering a linked list (Solution)

Similar to `[x for x in lst if f(x)]`

```python
def filter_link(f, ll):
    """Return a Link that contains only the elements x of Link LL for which f(x) is a true value.
    >>> is_odd = lambda x: x % 2 == 1
    >>> filter_link(is_odd, range_link(3, 6))
    Link(3, Link(5))
    """
    if ll is Link.empty:
        return Link.empty
    elif f(ll.first):
        return Link(ll.first, filter_link(f, ll.rest))
    return filter_link(f, ll.rest)
```

Try in PythonTutor
Mutating linked lists
Linked lists can change

Attribute assignments can change `first` and `rest` attributes of a `Link`.

```
s = Link("A", Link("B", Link("C")))
s.first = "Hi"
s.rest.first = "Hola"
s.rest.rest.first = "Oi"
```

Try in PythonTutor
Beware infinite lists

The rest of a linked list can contain the linked list as a sub-list.

```python
s = Link("A", Link("B", Link("C")))
t = s.rest
t.rest = s

s.first
s.rest.rest.rest.rest.rest.first
```
Exercise: Adding to front of linked list

```python
def insert_front(linked_list, new_val):
    """Inserts NEW_VAL in front of LINKED_LIST, returning new linked list."

    >>> ll = Link(1, Link(3, Link(5)))
    >>> insert_front(ll, 0)
    Link(0, Link(1, Link(3, Link(5))))
    """
```
Exercise: Adding to front of linked list (Solution)

```python
def insert_front(linked_list, new_val):
    """Inserts NEW_VAL in front of LINKED_LIST, returning new linked list."

    >>> ll = Link(1, Link(3, Link(5)))
    >>> insert_front(ll, 0)
    Link(0, Link(1, Link(3, Link(5))))
    """
    return Link(new_val, linked_list)
```
Exercise: Adding to an ordered linked list

```
def add(ordered_list, new_val):
    """Add NEW_VAL to ORDERED_LIST, returning modified ORDERED_LIST.
    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5))))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))))
    """
    if new_val < ordered_list.first:
        ... 
        return ordered_list

    elif new_val > ordered_list.first and ordered_list.rest is Link.empty:
        ... 

    elif new_val > ordered_list.first:
        ... 

    return ordered_list
```
Exercise: Adding to an ordered linked list (Solution)

```python
def add(ordered_list, new_val):
    """Add NEW_VAL to ORDERED_LIST, returning modified ORDERED_LIST."""
    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5)))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5)))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))))
    """
    if new_val < ordered_list.first:
        original_first = ordered_list.first
        ordered_list.first = new_val
        ordered_list.rest = Link(original_first, ordered_list.rest)
    elif new_val > ordered_list.first and ordered_list.rest is Link.empty:
        ordered_list.rest = Link(new_val)
    elif new_val > ordered_list.first:
        add(ordered_list.rest, new_val)
    return ordered_list
```

Insert value: 0 @ index: 0 ✅ Insert
Showdown: Python list vs. Link

The challenge:

- Store all the half-a-million words in "War and Peace"
- Insert a word at the beginning.

<table>
<thead>
<tr>
<th>Version</th>
<th>10,000 runs</th>
<th>100,000 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Link</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Try it yourself on your local machine (Legit Python!): warandpeace.py
Showdown: Python list vs. Link

The challenge:

- Store all the half-a-million words in "War and Peace"
- Insert a word at the beginning.

<table>
<thead>
<tr>
<th>Version</th>
<th>10,000 runs</th>
<th>100,000 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python list</td>
<td>2.6 seconds</td>
<td>37 seconds</td>
</tr>
<tr>
<td>Link</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Try it yourself on your local machine (Legit Python!): warandpeace.py
Showdown: Python list vs. Link

The challenge:

- Store all the half-a-million words in "War and Peace"
- Insert a word at the beginning.

<table>
<thead>
<tr>
<th>Version</th>
<th>10,000 runs</th>
<th>100,000 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python list</td>
<td>2.6 seconds</td>
<td>37 seconds</td>
</tr>
<tr>
<td>Link</td>
<td>0.01 seconds</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Try it yourself on your local machine (Legit Python!): warandpeace.py
Trees
Tree concepts

- A tree has a **root label** and a list of **branches**
- Each **branch** is itself a **tree**
- A tree with **zero branches** is called a **leaf**
Trees: Data abstraction

This is what we've been using:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree(label, branches)</td>
<td>Returns a tree with given LABEL at its root, whose branches are BRANCHES</td>
</tr>
<tr>
<td>label(tree)</td>
<td>Returns the label of root node of TREE</td>
</tr>
<tr>
<td>branches(tree)</td>
<td>Returns the branches of TREE (each a tree).</td>
</tr>
<tr>
<td>is_leaf(tree)</td>
<td>Returns true if TREE is a leaf node.</td>
</tr>
</tbody>
</table>
Trees: Data abstraction

Using an implementation like this:

```python
def tree(label, branches=[]):
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_leaf(tree):
    return not branches(tree)
```

How could we represent trees as a Python class?
A Tree class

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches

What's different? What's the same?
## tree versus Tree

<table>
<thead>
<tr>
<th>tree</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t = \text{tree}(\text{label, branches}=[]) )</td>
<td>( t = \text{Tree}(\text{label, branches}=[]) )</td>
</tr>
<tr>
<td>branches(( t ))</td>
<td>( t.\text{branches} )</td>
</tr>
<tr>
<td>label(( t ))</td>
<td>( t.\text{label} )</td>
</tr>
<tr>
<td>is_leaf(( t ))</td>
<td>( t.\text{is_leaf()} )</td>
</tr>
</tbody>
</table>

```python
def fib_tree(n):
    if n == 0 or n == 1:
        return tree(n)
    else:
        left = fib_tree(n - 2)
        right = fib_tree(n - 1)
        fib_n = label(left) + label(right)
        return tree(fib_n, [left, right])
```

```python
def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n - 2)
        right = fib_tree(n - 1)
        fib_n = left.label + right.label
        return Tree(fib_n, [left, right])
```
A fancier Tree

This is what assignments actually use:

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(self.label, branch_str)

    def __str__(self):
        return '
'.join(self.indented())

    def indented(self):
        lines = []
        for b in self.branches:
            for line in b.indented():
                lines.append(' ' + line)
        return [str(self.label)] + lines

It's built in to code.cs61a.org, and remember, you can `draw()` any tree/Tree.
Tree mutation
def double(t):
    """Doubles every label in T, mutating T."
    >>> t = Tree(1, [Tree(3, [Tree(5)]), Tree(7)])
    >>> double(t)
    >>> t
    Tree(2, [Tree(6, [Tree(10)]), Tree(14)])
    """
    t.label = t.label * 2
    for b in t.branches:
        double(b)
Exercise: Pruning trees

Removing subtrees from a tree is called **pruning**.

Always prune branches before recursive processing.

```python
def prune(t, n):
    """Prune all sub-trees whose label is n.
    >>> t = Tree(3, [Tree(1, [Tree(0), Tree(1)]), Tree(2, [Tree(1),
    >>> prune(t, 1)
    >>> t
    Tree(3, [Tree(2)])
    """
    t.branches = [___ for b in t.branches if ___]
    for b in t.branches:
        prune(___, ___)
```
Exercise: Pruning trees (Solution)

Removing subtrees from a tree is called **pruning**.

Always prune branches before recursive processing.

```python
def prune(t, n):
    """Prune all sub-trees whose label is n."
    >>> t = Tree(3, [Tree(1, [Tree(0), Tree(1)]), Tree(2, [Tree(1),
    >>> prune(t, 1)
    >>> t
    Tree(3, [Tree(2)])
    """
    t.branches = [b for b in t.branches if b.label != n]
    for b in t.branches:
        prune(b, n)
```
Recursive objects

Why are Tree and Link considered recursive objects?
Recursive objects

Why are Tree and Link considered recursive objects?

Each type of object contains references to the same type of object.

- An instance of Tree can contain additional instances of Tree, in the branches variable.
- An instance of Link can contain an additional instance of Link, in the rest variable.

Both classes lend themselves to recursive algorithms. Generally:

- For Tree: The base case is when is_leaf() is true; the recursive call is on the branches.
- For Link: The base case is when the rest is empty; the recursive call is on the rest.
Python Project of The Day!
Exotic Trees

A Field Guide to Exotic Trees: A gallery of trees programmatically generated from a Python script.

Github repository, Blog post